MULTIPLICATION: GROUP PROGRAMMING ASSIGNMENT - 1

Monica Bernard
COT 5405
PID: 4535013
Group 14

Pseudocode

1. Generate random numbers & store them in list1[0...n] & list2[0..n]
2. For num1 and num2 in (list1, list2)
   a. Result = multiply(num1, num2)
   b. Verify the product and add the product to the list result[]
   c. Print the size of result and total execution time

For num1 and num2 from list 1, list 2

```
1.   multiply(num1,num2)
2.   set result =0
3.   WHILE (y!=0) //check if Y has reduced to 0
4.   {
5.    IF Y is odd
6.     add x to result //call add function
7.   }
8.   double x by performing a left shift
9.   halve Y by performing a right shift
10.  return result
```

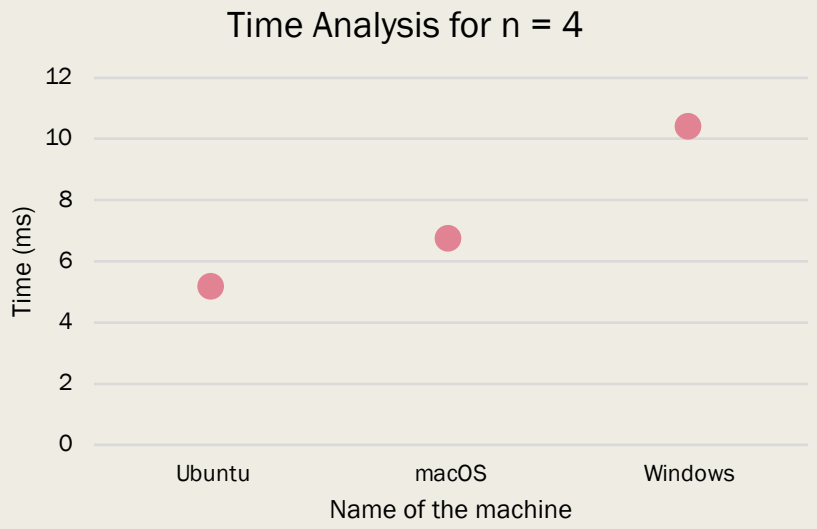
If y is odd in multiply, perform addition

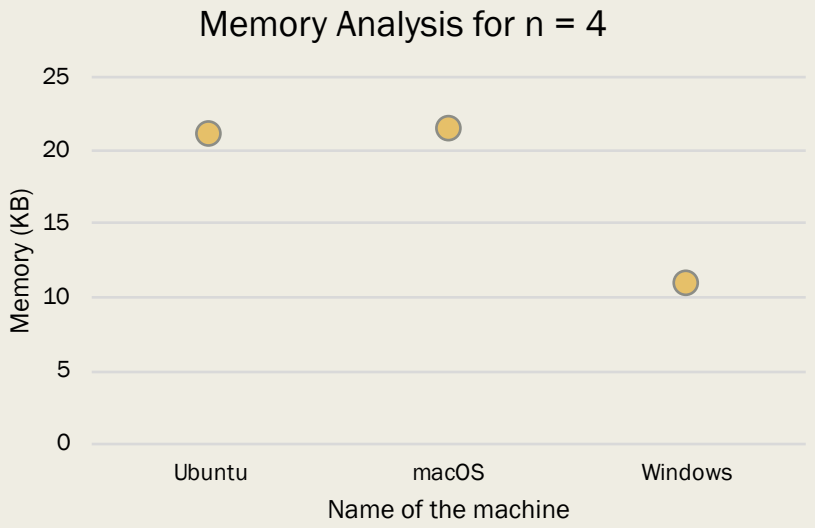
```
1.   add(x, y)
2.   WHILE (y != 0)
3.    {
4.     carry = x Logical_AND y
5.     x = ((x Logical_XOR y) % MASK)
6.     y = ((carry << 1) % MASK)
7.    }
8.   IF (x <= MAX_INT)
9.    return x
10.  ELSE
11.   //Number is negative
12.   return (~(x % MIN_INT) Logical_XOR MAX_INT)
```

Performance of the program on 1000 random inputs each of size n = 4 bits

- The following readings were obtained for our program with 64-bit Python on different machines:

Operating System	Number of cores	Frequency of the Processor (GHz)	Execution Time (ms)	Memory Utilization (bytes)
Ubuntu	4	3.4	5.1780	21016
macOS	4	2	6.7082	21272
Windows	4	3.4	10.3640	10728

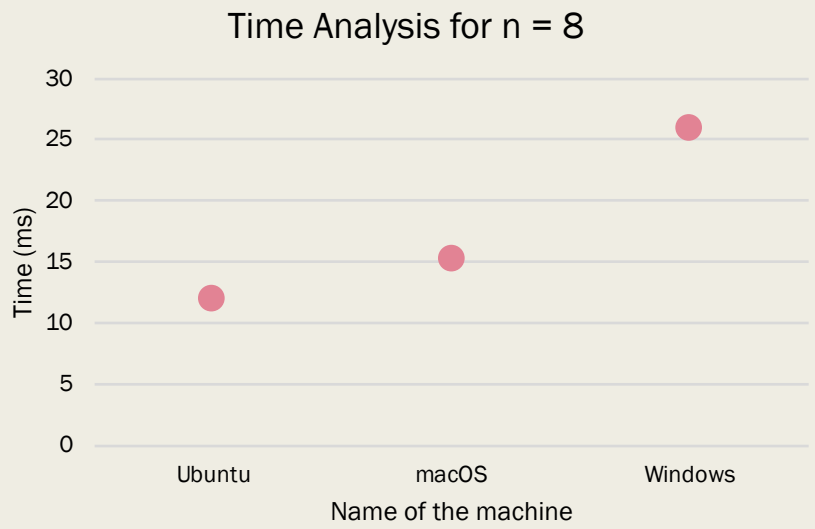


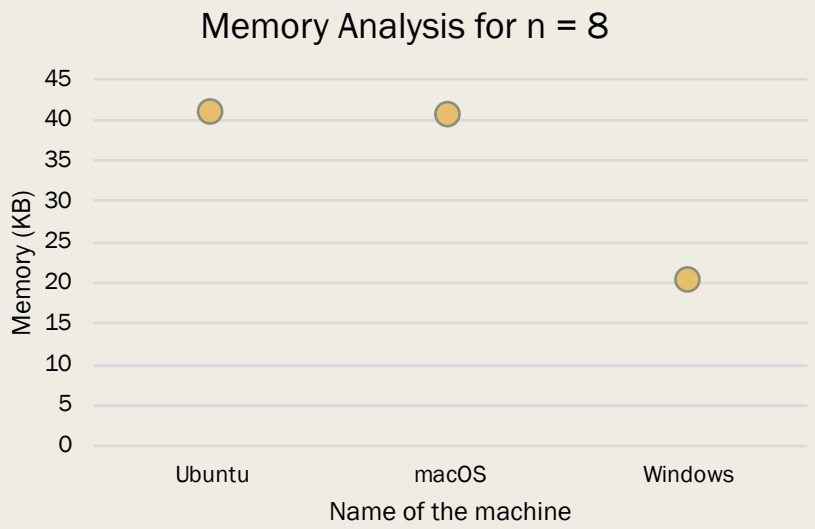


Performance of the program on 1000 random inputs each of size n = 8 bits

- The following readings were obtained for our program with 64-bit Python on different machines:

Operating System	Number of cores	Frequency of the Processor (GHz)	Execution Time (ms)	Memory Utilization (bytes)
Ubuntu	4	3.4	11.9250	40632
macOS	4	2	15.0767	40504
Windows	4	3.4	25.9314	20296

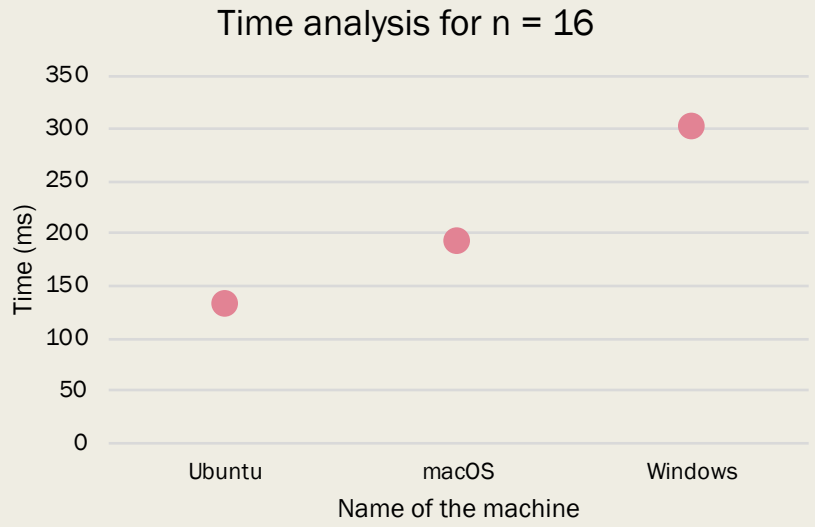


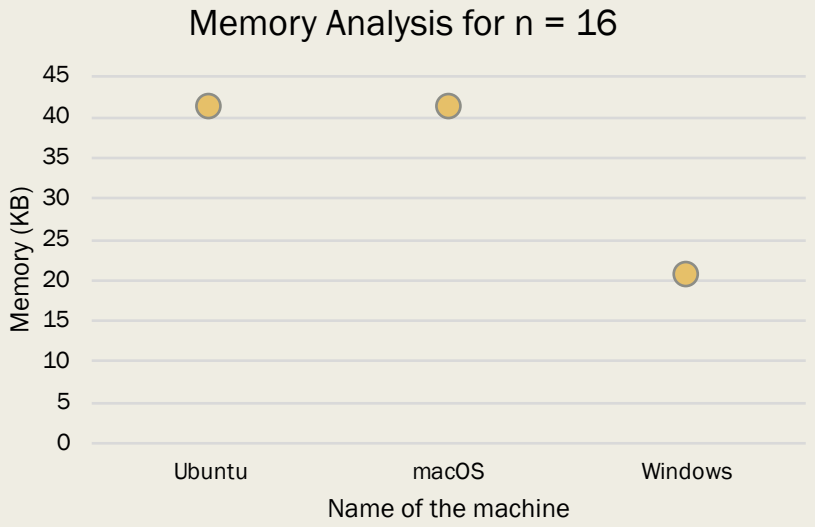


Performance of the program on 1000 random inputs each of size n = 16 bits

- The following readings were obtained for our program with 64-bit Python on different machines:

Operating System	Number of cores	Frequency of the Processor (GHz)	Execution Time (ms)	Memory Utilization (bytes)
Ubuntu	4	3.4	25.8010	41024
macOS	4	2	38.6596	41024
Windows	4	3.4	58.3064	20520

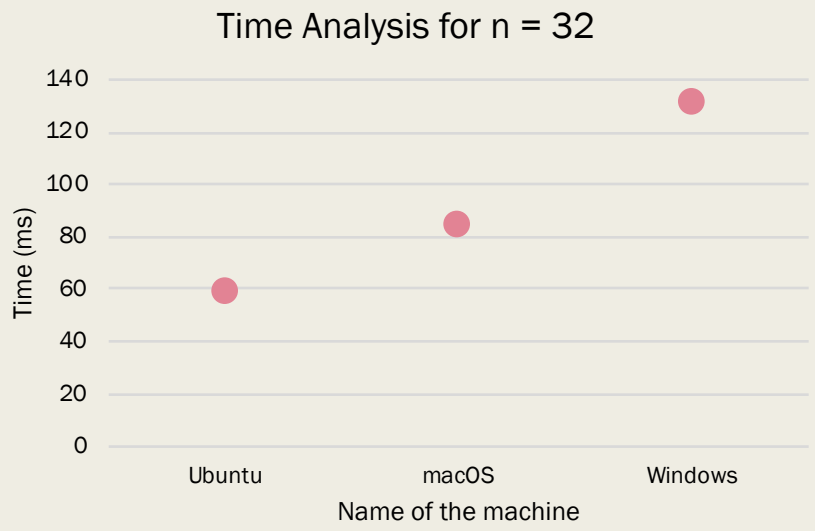


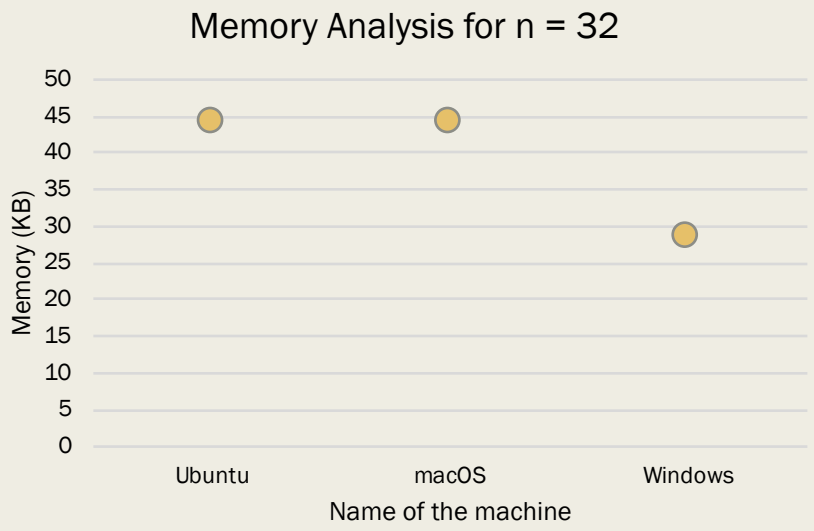


Performance of the program on 1000 random inputs each of size n = 32 bits

- The following readings were obtained for our program with 64-bit Python on different machines:

Operating System	Number of cores	Frequency of the Processor (GHz)	Execution Time (ms)	Memory Utilization (bytes)
Ubuntu	4	3.4	5.1780	25.8010
macOS	4	2	6.7082	84.2128
Windows	4	3.4	10.3640	130.8258

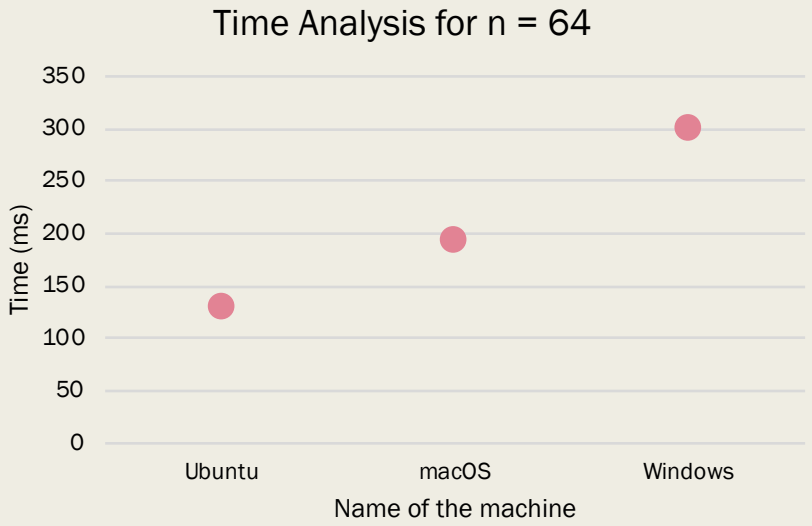


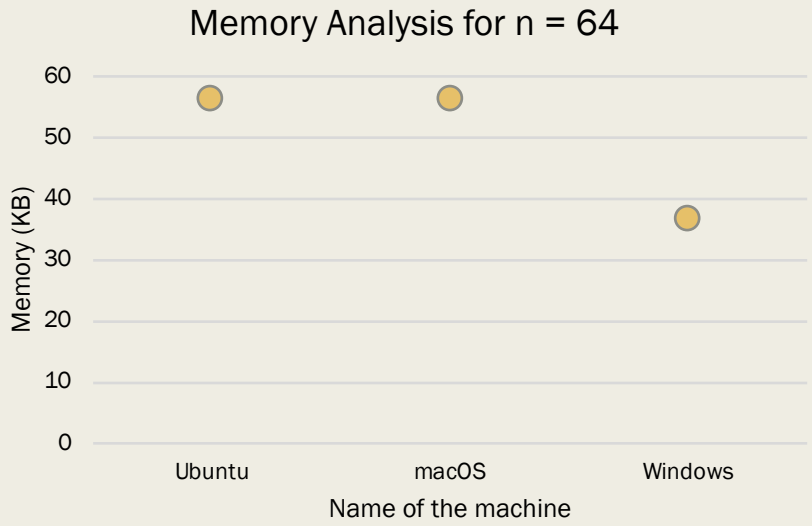


Performance of the program on 1000 random inputs each of size n = 64 bits

- The following readings were obtained for our program with 64-bit Python on different machines:

Operating System	Number of cores	Frequency of the Processor (GHz)	Execution Time (ms)	Memory Utilization (bytes)
Ubuntu	4	3.4	130.2690	36520
macOS	4	2	192.8478	56344
Windows	4	3.4	301.6521	56360

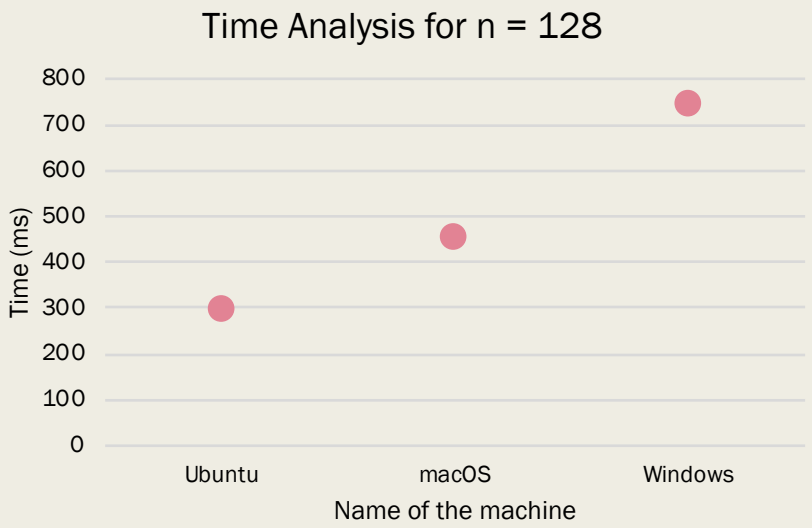


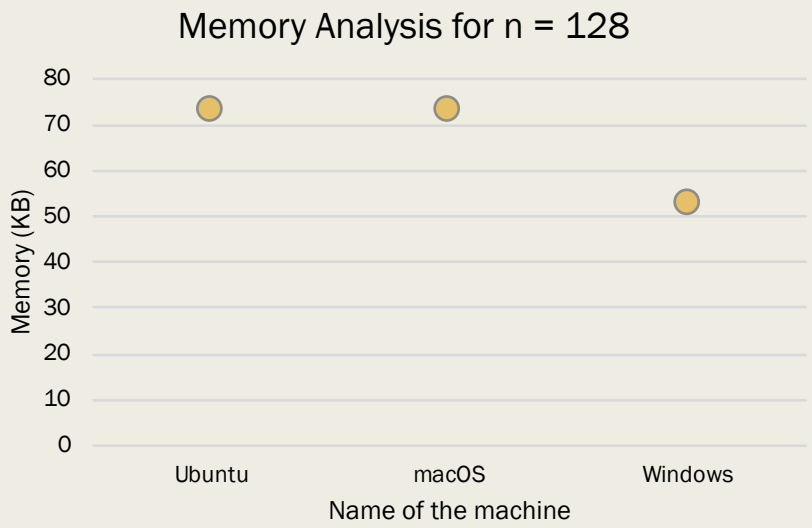


Performance of the program on 1000 random inputs each of size n = 128 bits

- The following readings were obtained for our program with 64-bit Python on different machines:

Operating System	Number of cores	Frequency of the Processor (GHz)	Execution Time (ms)	Memory Utilization (bytes)
Ubuntu	4	3.4	294.5530	73008
macOS	4	2	449.4062	73024
Windows	4	3.4	743.0900	52520

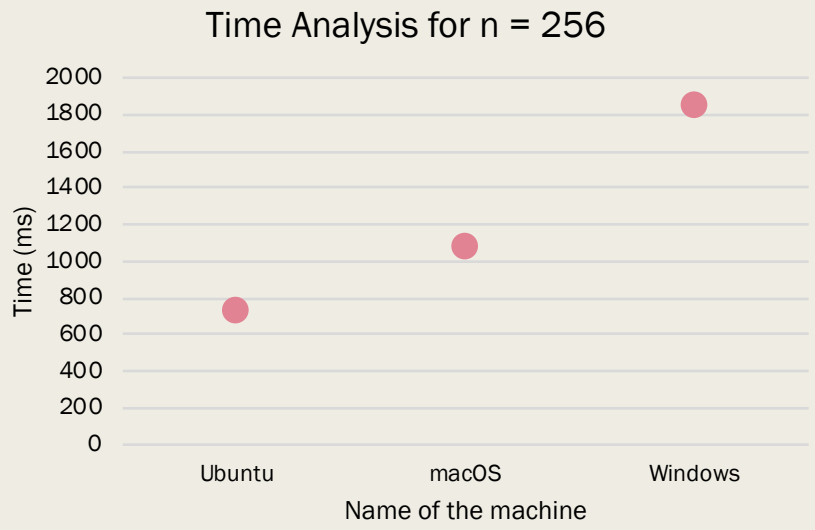


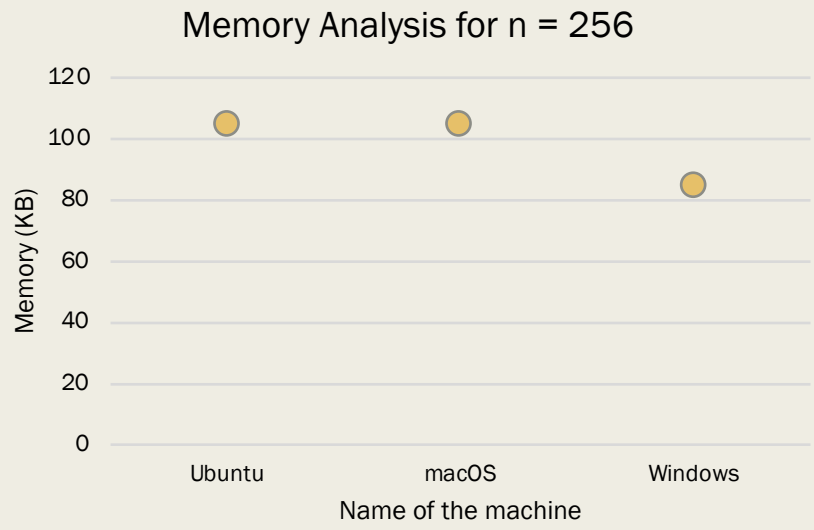


Performance of the program on 1000 random inputs each of size n = 256 bits

- The following readings were obtained for our program with 64-bit Python on different machines:

Operating System	Number of cores	Frequency of the Processor (GHz)	Execution Time (ms)	Memory Utilization (bytes)
Ubuntu	4	3.4	725.1330	105024
macOS	4	2	1078.5573	105024
Windows	4	3.4	1846.0271	84520

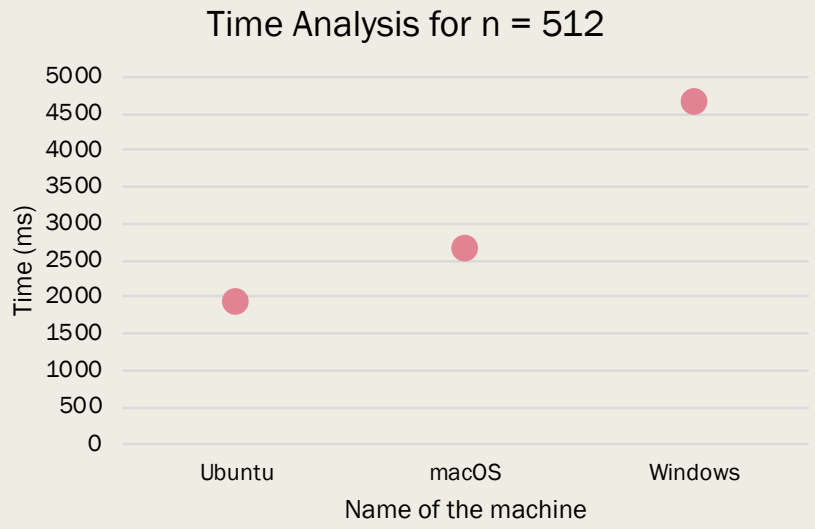


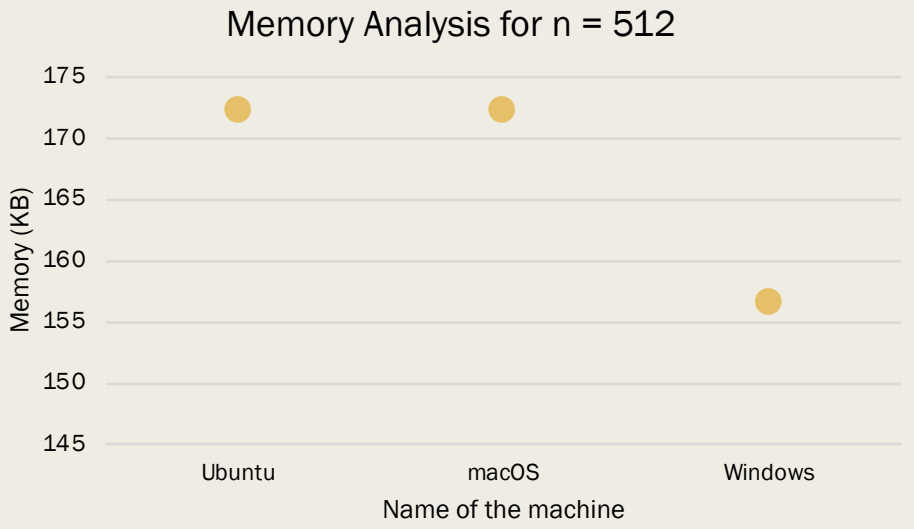


Performance of the program on 1000 random inputs each of size n = 512 bits

- The following readings were obtained for our program with 64-bit Python on different machines:

Operating System	Number of cores	Frequency of the Processor (GHz)	Execution Time (ms)	Memory Utilization (bytes)
Ubuntu	4	3.4	1931.7090	172224
macOS	4	2	2639.0396	172208
Windows	4	3.4	4668.2208	156520

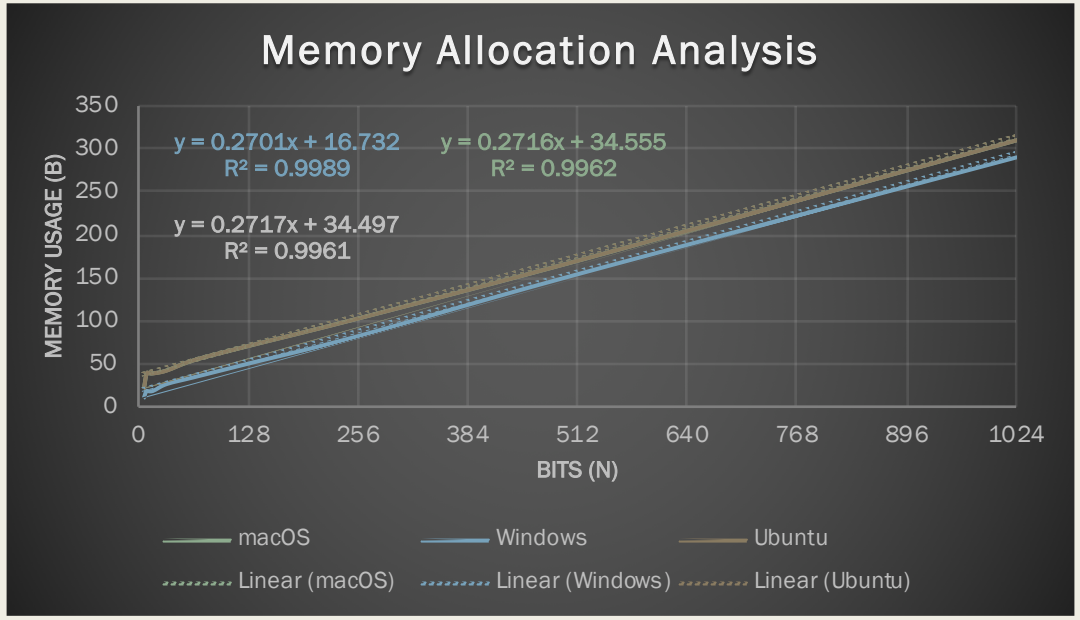


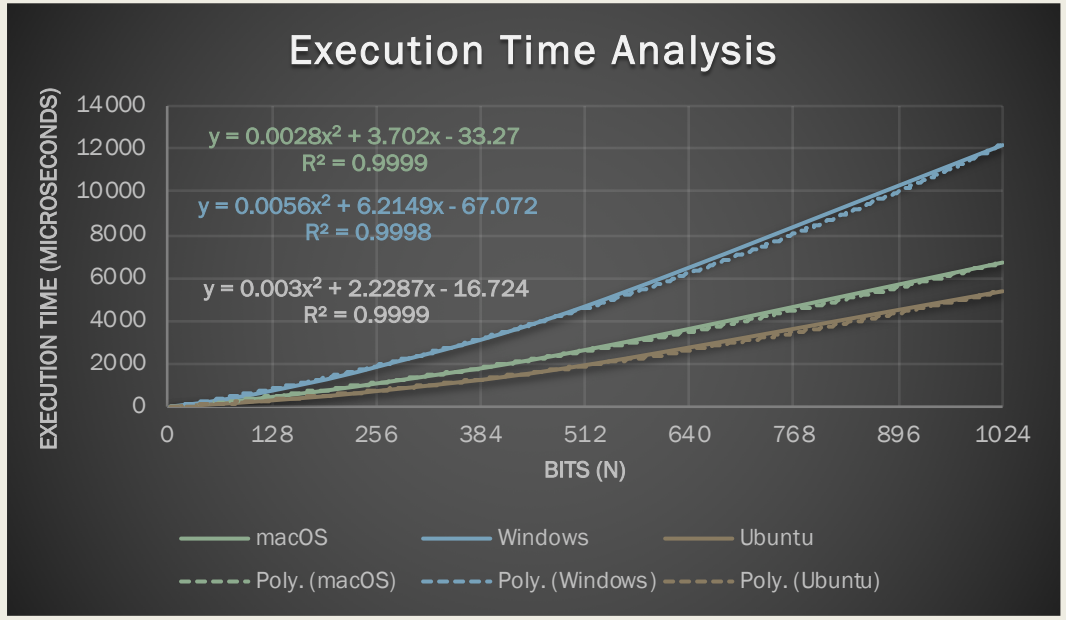


Average performance on inputs of various sizes and conjecture a time/space complexity for the implementation

	Memory (Bytes)		
Bits	macOS	Windows	Ubuntu
4	21.272	10.728	21.016
8	40.504	20.296	40.632
16	41.024	20.52	41.024
32	44.36	28.52	44.16
64	56.344	36.52	56.36
128	73.024	52.52	73.008
256	105.024	84.52	105.024
512	172.208	156.52	172.224
1024	312.32	292.52	312.384

	Time (μs)		
Bits	macOS	Windows	Ubuntu
4	6.7082	10.364	5.178
8	15.0767	25.9314	11.925
16	38.6596	58.364	25.801
32	84.2128	130.8258	58.845
64	192.8478	301.6521	130.269
128	449.4062	743.09	294.553
256	1078.5573	1846.0271	725.133
512	2639.0396	4668.2208	1931.709
1024	6713.2664	12181.7364	5376.48





Time/Space Complexity based on the Pseudocode:

```
1.  multiply(num1,num2)
2.  set result =0
3.  WHILE (y!=0) //check if Y has
    reduced to 0                    ⇒ n + 1
4.  {
5.   IF Y is odd                    ⇒ n
6.    //call add function
7.      Add function has another
        while loop in it            ⇒ n * n
8.  }
9.  Double x by performing a left
    shift                           ⇒ n
10. halve Y by performing a right
    shift                           ⇒ n
11. return result
```

- The algorithm used to multiply any two numbers is called the Russian Peasant's multiplication. The complexity of the algorithm depends on the number of times while loops get executed.
- The while loop in the multiply function (line 1) gets executed n + 1 number of times. Once the execution enters the while loop, it checks the 'if' condition which is executed n times.
- Inside this while loop, we have another while loop from the add function (line 5). Hence, the total number of times the nested while loop gets executed = n * n = n^2.
- Both the bit shift operations in line 6 and 7 run for n times each.
- Hence the time complexity is (n + 1) + n + n^2 + n + n = n^2 + 4n + 1, which belongs to **O(n^2)**.
- The final result for the multiplication is stored in a list called 'result'. The size of this list increases linearly with the size of the input elements. Hence, the space complexity is **O(n)**.
- The theoretical calculation of time and space complexity using Big-O notation is in line with the results obtained from the execution of our program.

GitHub link for the code:

- https://github.com/monicabernard/COT-5405_project1/blob/master/multiply_final.py

Thank you.

Monica Bernard